Wzorce projektowe
Opracowanie

Spis treści

1 Na podstawie

- https://www.journaldev.com/1827/java-design-patterns-example-tutorial
- Książka - Designed Patterns exaplined simply
- Książka - Head first Design Patterns
- wzorce projektowe, opracowanie 2019 z wiki
- https://refactoring.guru/design-patterns/chain-of-responsibility

2 Podział wzorców projektowych

Pytanie

Na jakie rodzaje dzielimy wzorce projektowe ? ?

Wzorce projektowe dzielimy na:

- Kreacyjne - opisują, w jaki sposób obiekty są tworzone, zapewniają sposoby na instancjację obiektów w najlepszy możliwy sposób w danej sytuacji
- Behawioralne - opisują zachowanie obiektów, w jaki sposób obiekty komunikują się ze sobą
- Strukturalne - opisują sposób, w jaki obiekty są zbudowane, definiują jak klasy i interfejsy mają być zbudowane w celu realizacji pewnych działań
- Architektoniczne - opisują oprogramowanie na bardziej abstrakcyjnym poziomie, skupiają się na architekturze rozwiązań raczej niż na ich szczegółach

3 Kontekst kolokwium

Na kolokwium należy dla danego wzorca podać:

- Charakterystyka
- Zastosowanie
- UML
- Kod
- Zalety i wady
- Porównanie z innymi wzorcami

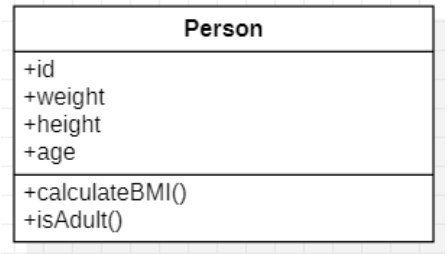
4 Podstawy diagramów klas UML

Przy rozważaniach o wzorcach często stosujemy notację UML w celu przedstawienia jak dany wzorzec jest implementowany, konieczne jest więc abyśmy poznali przynajmniej jakieś podstawy tej notacji. Ograniczymy się do niezbędnego minimum z diagramów klas.

Podstawowe komponenty z których budujemy nasz diagram klas to klasy i interfejsy, które rozumiemy w takim sensie jak są rozumiane w językach programowania.

4.1. Klasa

Klasa składa się z trzech obszarów, pierwszy z nich jest przeznaczony na nazwę klasy, środkowy to miejsce na atrybuty klasy, a na dole umieszczamy metody klasy. Przykładowo:



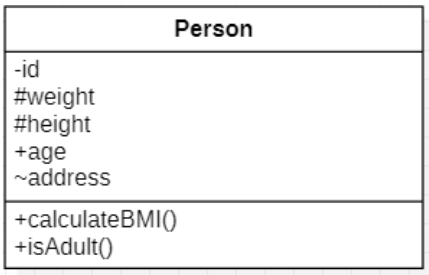
Rysunek 1: Przykładowa reprezentacja klasy w UML

4.1.1. Widoczność atrybutów klasy

Na diagramie klas możemy uwzględnić widoczność atrybutów i metod (private, protected, itd.). Widoczność ta jest oznaczana symbolem poprzedzającym nazwę atrybutu w następujący sposób:

- + (public)
- - (private)
- # (protected)
- ~(default, package)
- / (do reprezentacji atrybutu, który został odziedziczony)

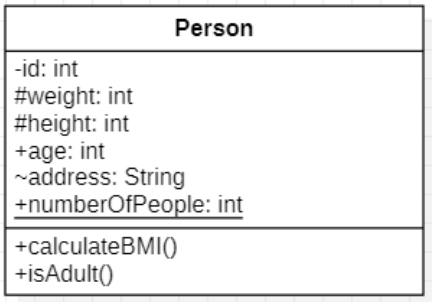
Pozmieniajmy więc widoczność paru pól w naszej klasie Person:



Rysunek 2: Uwzględnienie widoczności atrybutów

4.1.2. Atrybuty statyczne i typy atrybutów

Na diagramie klas możemy także uwzględnić atrybuty statyczne, reprezentujemy je przez podkreślenie atrybutu, a także typy atrybutów, te reprezentujemy po nazwie atrybutu i dwukropku. Typy mogą się oczywiście różnić pomiędzy językami, możemy wybrać ten w którym aktualnie kodujemy lub przyjąć jakieś ogólnie rozumiane typy. Dodajemy więc atrybut statyczny oraz typy do naszej klasy Person.

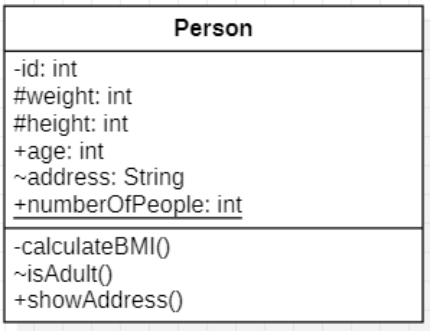


Rysunek 3: Uwzględnienie atrybutów statycznych oraz typów atrybutów.

4.1.3. Widoczność metod klasy

W analogiczny sposób do atrybutów możemy dodawać informacje o widoczności do naszych metod:

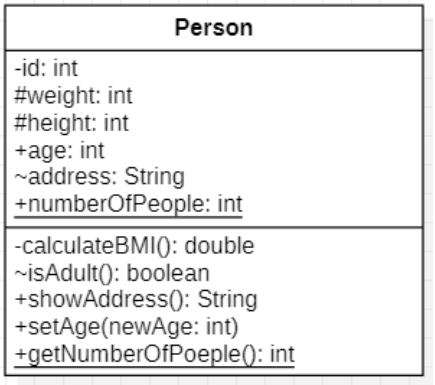
- + (public)
- - (private)
- # (protected)
- ~(default, package)



Rysunek 4: Uwzględnienie atrybutów statycznych oraz typów atrybutów.

4.1.4. Metody statyczne, typy argumentów i typ zwracany

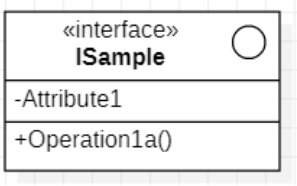
Analogicznie jak dla atrybutów możemy oznaczyć w klasie metodę statyczną poprzez jej podkreślenie. Typy argumentów wejściowych zaznaczamy tak samo jak typy atrybutów, tj. po dwukropku. Typ zwracany przez metodę zapisujemy po dwukropku po nazwie metody, jeśli typem tym jest void to nic nie zapisujemy. Dodajmy więc nowe możliwości do naszej klasy Person:



Rysunek 5: Uwzględnienie metod statycznych, typów argumentów i typu zwracanego

4.2. Interfejsy

Interfejsy różnią się od klas jedynie tym, że w pierwszej sekcji zawierają dodatkowo przed nazwą wyrażenie «interface» oznaczające, że dany komponent jest interfejsem. Poza tym, metody i atrybuty definiujemy analogicznie jak dla klas. W programie STAR UML interfejs ma dodatkowo w reprezentacji graficznej kółko jak widać na obrazku poniżej, jednak w ogólności nie jest ono zawierane w interfejsach na diagramach UML.



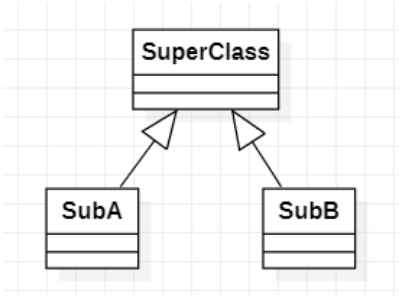
Rysunek 6: Przykład interfejsu

4.3. Relacje

Relacje służą nam do opisu zależności pomiędzy interfejsami i klasami. Możemy wyrażać wszystkie relacje, które typowo stosujemy w programowaniu, tj. agregacje, implementacje, dependencje itd.

4.3.1. Dziedziczenie (generalizacja)

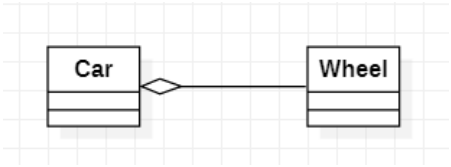
Dziedziczenie jest relacją, którą znamy z programowania obiektowego. Na diagramie UML jest reprezentowane strzałką w pustym grotem, strzałka zawsze wskazuje na superklase.



Rysunek 7: Relacja dziedziczenia na diagramie UML

4.3.2. Agregacja

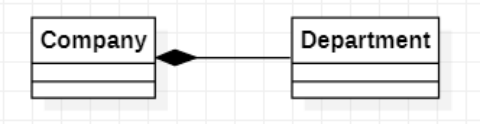
To relacja, która informuje nas, że dana klasa jest częścią innej klasy, ale również może istnieć niezależnie od tej klasy. Przykładowo koło od samochodu może istnieć zanim jeszcze samochód zostanie wyprodukowany. Można więc powiedzieć, że koło istnieje niezależnie od samochodu, pomimo, że każdy samochód ma koła. Taką relacje modelujemy przy pomocy agregacji, czyli strzałki z grotem w kształcie pustego rombu. Grot ten jest zawsze skierowany w stronę klasy, która zawiera inne klasy.



Rysunek 8: Przykład agregacji

4.3.3. Kompozycja

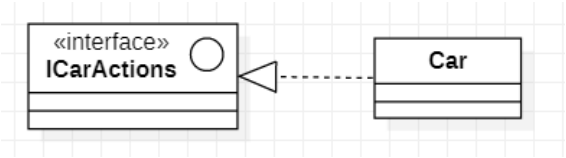
To relacja bardzo podobna do agregacji z tą różnicą, że obiekt który jest częścią innego nie może istnieć niezależnie, tj. jeśli obiekt, który zawiera inne obiekty zostaje zniszczony, również wszystkie obiekty w nim zawarte zostają zniszczone. Dodatkowo, każdy z obiektów stanowiących części innego ma przypisany dokładnie jeden taki obiekt. Przykładem takiej relacji może być firma i jej działy. Każda firma ma wiele działów, jednak nie ma sensu mówić o działach bez kontekstu firmy. Relację taką reprezentujemy tak jak relację agregacji z tą różnicą, że tym razem romb jest wypełniony.



Rysunek 9: Przykład kompozycji

4.3.4. Realizacja

To relacja, która odpowiada relacji implementacji interfejsu jaką znamy z języków programowania. Relacja tą oznaczamy strzałką przerywaną z pustym grotem (podobnie jak dziedziczenie, ale tutaj strzałka jest przerywana, a nie ciągła). Strzałka zawsze skierowana jest w stronę interfejsu, który implementujemy.



Rysunek 10: Przykład realizacji

4.3.5. Asocjacja

To relacja, która informuje nas, że dana klasa korzysta z funkcjonalności innej klasy lub komunikuje się z nią. Może ona być jednokierunkowa (strzałka z cienkim grotem) lub dwukierunkowa (prosta linia). Asocjacja może też wyrażać, że dana klasa zawiera inną klasę (tak jak agregacja czy kompozycja). Jaka jest więc różnica pomiędzy asocjacją, a asocjacją i kompozycją ? Otóż asocjacja jest terminem bardziej ogólnym, może wyrażać więcej niż tylko zawieranie, ale także komunikacje, korzystanie z funkcjonalności itp. Dlatego do zawierania się klas w innych raczej należy stosować agregację lub kompozycje.

Rysunek 11: Asocjacja dwukierunkowa

5 Wzorce Behawioralne

5.1. Obserwator (Observer)

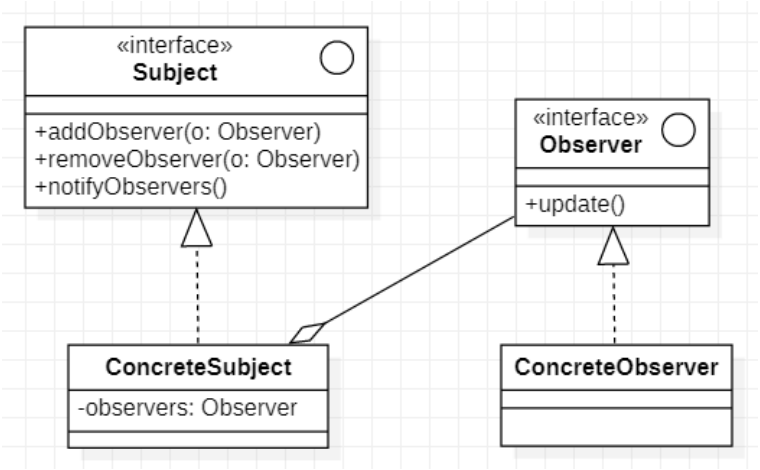
5.1.1. Charakterystyka

Obserwator to behawioralny wzorzec projektowy, który pozwala zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach zachodzących w obiekcie, który te obiekty obserwują.

5.1.2. Zastosowanie

- Kiedy należy powiadomić zmieniającą się listę obiektów o pewnym zdarzeniu, np. naciśnięciu elementu GUI
- Gdy zmiany stanu jednego obiektu mogą wymagać zmiany innych obiektów, a rzeczywisty zestaw obiektów jest wcześniej nieznany lub zmienia się dynamicznie
- Gdy niektóre obiekty w aplikacji muszą obserwować inne, ale tylko przez ograniczony czas lub w określonych przypadkach.

5.1.3. UML



Rysunek 12: Asocjacja dwukierunkowa

5.1.4. Kod

Github.

5.1.5. Zalety i wady

- Zalety
  - Możesz wprowadzić nowe klasy subskrybentów bez konieczności zmiany kodu wydawcy (i na odwrót, jeśli istnieje interfejs wydawcy).
  - Możesz ustanowić relacje miedzy obiektami w czasie wykonywania.

- Wady
  - Subskrybenci są powiadamiani w kolejności losowej
  - Ciężko jest śledzić flow aplikacji

5.1.6. Porównanie do innych wzorców

Łańcuch odpowiedzialności, dowództwo, mediator i obserwator poruszają różne sposoby łączenia nadawców i odbiorców wniosków:

- Łańcuch Odpowiedzialności przekazuje zadanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, dopóki jeden z nich nie zajmie się nim, w przeciwieństwie do Observera wzorzec ten nie wymaga zawierani w sobie referencji klas komunikujących się, podobnie jak observer wzorzec ten może być modyfikowany w czasie wykonywania programu (modyfikacja handlerów i ich kolejności)
- Command ustanawia jednokierunkowe połaczenia miedzy nadawcami i odbiorcami
- Mediator eliminuje bezpośrednie połączenia miedzy nadawcami i odbiorcami, zmuszając ich do pośredniego komunikowania się za pośrednictwem obiektu mediatora
- Obserwator pozwala odbiorcom dynamicznie subskrybować i rezygnować z otrzymywania zadań

Różnica miedzy Mediatorem a Obserwatorem jest często nieuchwytna. W większości przypadków można wdrożyc jeden z tych wzorów, ale czasami można zastosować oba jednocześnie.

5.2. Chain of responsibility

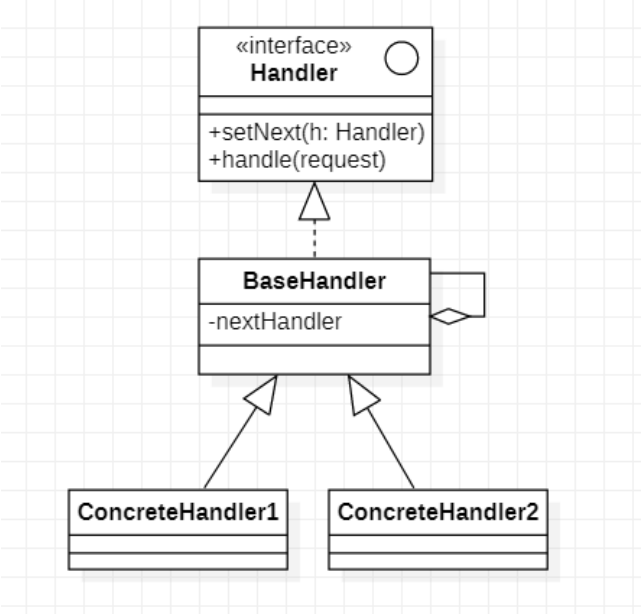
5.2.1. Charakterystyka

Pozwala przekazywać zadania wzdłuż łańcucha handlerów. Po otrzymaniu zadania, każdy handler postanawia albo przetworzyć zadanie, albo przekazać je do następnego handlera w łańcuchu.

5.2.2. Zastosowanie

- Obsługa żądań, które wymagają walidacji danych i autoryzacji
- Obsługa zdarzeń w GUI, przechodząc po drzewie elementów GUI szukamy elementu mogącego obsłużyć zdarzenie
- Wszystkie sytuacje gdzie konieczne jest wykonanie kilku procedur obsługi w określonej kolejności

5.2.3. UML



Rysunek 13: Wzorzec Chain of responsibility

5.2.4. Kod

Github

5.2.5. Zalety i wady

- Zalety
  - Rozdzielenie klas wysyłających komunikaty od odbierających komunikaty
  - Możliwość modyfikacji bez naruszenia istniejącego kodu
  - Możliwość zmian w czasie wykonywania programu (np. dodanie nowego handlera, zmiana kolejności)
- Wady
  - Brak gwarancji obsługi żądania przez łańcuch

5.2.6. Porównanie do innych wzorców

Łańcuch odpowiedzialności, dowództwo, mediator i obserwator poruszają różne sposoby łączenia nadawców i odbiorców wniosków:

- Łańcuch Odpowiedzialności przekazuje zadanie sekwencyjnie wzdłuż dynamicznego łańcucha potencjalnych odbiorców, dopóki jeden z nich nie zajmie się nim, w przeciwieństwie do Observera wzorzec ten nie wymaga zawierani w sobie referencji klas komunikujących się, podobnie jak observer wzorzec ten może być modyfikowany w czasie wykonywania programu (modyfikacja handlerów i ich kolejności)

- Command ustanawia jednokierunkowe połaczenia miedzy nadawcami i odbiorcami
- Mediator eliminuje bezpośrednie połączenia miedzy nadawcami i odbiorcami, zmuszając ich do pośredniego komunikowania się za pośrednictwem obiektu mediatora
- Obserwator pozwala odbiorcom dynamicznie subskrybować i rezygnować z otrzymywania zadań
- Łańcuch odpowiedzialności jest często używany w połączeniu z Composite. W takim przypadku, gdy komponent typu liść otrzyma zadanie, może przekazać go przez łańcuch wszystkich komponentów nadrzędnych do katalogu głównego drzewa obiektów.
- Handleri w łańcuchu odpowiedzialności mogą być implementowane jako Command. W takim przypadku można wykonać wiele różnych operacji w tym samym obiekcie kontekstu, reprezentowanym przez zadanie.
- Łańcuch odpowiedzialności i dekorator maja bardzo podobne struktury klasowe. Oba wzorce polegają na rekurencyjnej kompozycji, aby przekazać wykonanie przez szereg obiektów. Istnieje jednak kilka istotnych różnic.